Visit CramSession.com

Comprehensive **Study Guides**

INFORMATION TECHNOLOGY

A+
Adobe
C++
Cisco CCNA

Your Trusted Study Resource for Technical Certifications

Written by experts.
The most popular study guides on the web.

In Versatile PDF file format

Check out these great features at www.cramsession.com

> Discussion Boards
http://boards.cramsession.com

> Info Center
http://infocenter.cramsession.com

> SkillDrill
http://www.skilldrill.com

> Newsletters
http://newsletters.cramsession.com/default.asp

> CramChallenge Questions
http://newsletters.cramsession.com/signup/default.asp#cramchallenge

> Discounts & Freebies
http://newsletters.cramsession.com/signup/ProdInfo.asp

Microsoft Office
Microsoft Windows 2000
Microsoft Windows XP
Network Security
Network+
Networking
Nortel Networks
Novell
Oracle
Proxy Server
Red Hat Linux
SAIR Linux
SANS
SCO
Server+
SQL
Sun Solaris
Unix
Visual Basic
Web Design

Sun Certified Java Programmer (Java 2)

Version 3.0.0

Notice: While every precaution has been taken in the preparation of this material, neither the author nor Cramsession.com assumes any liability in the event of loss or damage directly or indirectly caused by any inaccuracies or incompleteness of the material contained in this document. The information in this document is provided and distributed "as-is", without any expressed or implied warranty. Your use of the information in this document is solely at your own risk, and Cramsession.com cannot be held liable for any damages incurred through the use of this material. The use of product names in this work is for information purposes only, and does not constitute an endorsement by, or affiliation with Cramsession.com. Product names used in this work may be registered trademarks of their manufacturers. This document is protected under US and international copyright laws and is intended for individual, personal use only.
For more details, visit our legal page.

Sun Certified Java Programmer (Java 2)

Version 3.0.0

NOTICE: **Got the NEWest Version?** Make sure by clicking here!

Abstract:

This study guide will help you to prepare for Sun exam 310-025, Sun Certified Java Programmer (Java 2). Exam topics include Declarations and Access Control, Flow Control and Exception Handling, Garbage Collection, Language Fundamentals, Operators and Assignments, Overloading, Overriding, Runtime Type and Object Orientation, and Java Packages and Layout.

Find even more help here:

> **Feedback & Discussion Board for this exam**

> Read & Write Reviews of this study guide

> Rate this Cramsession study guide

© 2001 All Rights Reserved – BrainBuzz.com

Contents:

Fundamentals of the Java Language 5
  Source Files 5
  Packages 5
  Import Statements 5
  The main() Method 5
  Identifiers 6
  Keywords 6
  Primitive Data Types 7
  Automatic Initalizations 8
Operators and Assignments 9
  Urnary Arithmetic Operators 9
  Binary Arithmetic Operators 9
    Numeric promotion 9
    Division 10
    Modulo 10
    Runtime Exceptions 10
    Overflow and Underflow 10
  String Operators 10
  Equality Operators 10
  Logical Operators 11
  Assignment Operators 12
  The Cast Operator 13
  Bitwise Operators 14
  Shift Operators 14
  The instanceof Operator 14
Declarations & Access Control 15
  Declaring Field and Local Variables 15

© 2001 All Rights Reserved – BrainBuzz.com

Declaring Arrays 15
Method Declarations 16
Access Modifiers 17
public 17
private 17
protected 17
(none) 17
Other Modifiers 18
static 18
final 18
abstract 18
native 18
synchronized 18
transient 18
volatile (not covered on exam) 19
Summary of Access Modifiers 19
Flow Control 19
Exception Handling 21
Overloading, Overriding, Runtime Type and Object Orientation 22
Object Orientation 22
Encapsulation 22
Polymorphism 22
Designing Classes with "is a" and "has a" Relationships 23
Overloading Methods 23
Overridden Methods 23
Constructors 23
Inner Classes 24
Anonymous Classes 24
Garbage Collection 25

© 2001 All Rights Reserved – BrainBuzz.com

Threads 25
The java.lang package 27
The Math class 28
The String Class 28
StringBuffer Class 29
The java.io Package 29
File Class 29
InputStreamReader and OutputStreamWriter Classes 30
FilterInputStream and FilterOutputStream...31
FileInputStream and FileOutputStream 31
RandomAccessFile Class 32
The java.util Package...32
The java.awt Package...34
Layout Managers 35

© 2001 All Rights Reserved – BrainBuzz.com

Fundamentals of the Java Language

Source Files

Each source file may only contain one *public* class or interface at most.

- The source file name must be the name of the public class or interface, followed by the `.java` extension.
- The order for every source file "heading" is as follows:

  package declarations

  import statements

  class definitions

Packages

- Java classes and interfaces are organized into **packages**.
- A **package** statement must appear as the first statement of a source file.
  ```
  package nameofpackage;
  ```
- If no package name is used, all of the classes and interfaces are put into the default no-name package.

Import Statements

The **import** statement is a way to reference classes and interfaces that are declared in other packages. There are two forms of using the import statement:

```
import nameofpackage.classname;   // Allows classname to be referenced
                                  without specifying package name.
import nameofpackage.*;           // Allows ALL classes and interfaces
                                  of nameofpackage to be referenced
                                  without specifying package name.
```

Note: The java.lang package is imported by default, so you do not have to import it explicitly.

The main() Method

- Serves as the main entry point for the application. All applications must have a main() method.
- It must be: public, static and void.
- It may **not** return a value - void return type.
- Has one argument - an array of strings, and it does not have to be *args*.

© 2001 All Rights Reserved – BrainBuzz.com

- It may take any of the following forms:

```
String[] args
String []arghs
String rags[]
```

- The order of public and static does not matter, but the keyword void must come right before main().

```
public static void main(String[] args)

static public void main(String args[])
```

- The args array is used to access the program's command line arguments. Arguments are placed in the String array starting at 0, after the java command and the program name. It would be invoked as follows:

`java testprogram 5 99`	The String object "5" would be accessed as args[0], and String object "99" would be accessed as args[1].

- Applets are not required to have a main() method.

Identifiers

- Java uses identifiers to name language entities. They may begin with a Unicode letter, underscore(_), or a dollar sign($).
- Identifiers ARE case sensitive.
- Identifiers cannot be the same as a reserved keyword.

Keywords

The following words are reserved by Java:

abstract	double	int	strictfp
boolean	else	interface	super
break	extends	long	switch
Byte	final	native	synchronized
Case	finally	new	this
Catch	float	package	throw
Char	for	private	throws

© 2001 All Rights Reserved – BrainBuzz.com

Class	goto	protected	transient
const	if	public	try
continue	implements	return	void
default	import	short	volatile
Do	instanceof	static	while

Note: null, true and false are literals, not keywords. Nonetheless, they should not be used as names in a program.

Primitive Data Types

- There are 8 primitives in Java.
- Remember that primitive variables only store primitive values – they are NOT references to objects.
- They occupy a pre-defined numbers of bits.

Type	Size	Range of Values
boolean	8 bits	true and false
byte	8 bit signed integer	-2^7 to 2^7-1
char	16 bit unsigned integer	0 to 2^{16}-1 '\u0000' to '\uffff'
short	16 bit signed integer	-2^{15} to 2^{15}-1
int	32 bit signed integer	-2^{31} to 2^{31}-1
long	64 bit signed integer	-2^{63} to 2^{63}-1
Float	32 bit floating point values	Float.MIN_VALUE to Float.MAX_VALUE, Float.NaN, Float.NEGATIVE_INFINITY, Float.POSITIVE_INFINITY
double	64 bit floating point values	Double.MAX_VALUE to Double,MIN_VALUE, Double.NaN, Double.NEGATIVE_INFINITY, Double.NEGATIVE

***Note**: For the exam, you do NOT have to know the ranges for floating point numbers.

- long - can be denoted by a trailing "l" or "L". e.g. 200L
- float - can be denoted by a trailing "f" or "F". e.g. 20.7f
- double - can be denoted by a trailing "d" or "D". e.g. 300D

© 2001 All Rights Reserved – BrainBuzz.com

- The boolean type may only have the values, true and false.
- String literals are a shorthand way to represent String objects, and so are not primitives. They are represented with characters enclosed with double quotes (").
- The backslash (\) is Java's escape code character.

Example \r represents a carriage return.
\' represents a single quote.

- You can represent Unicode characters by using a \u, followed by four hexadecimal digits (\u0000 through \uffff).

Example \u1234

- When an object is not assigned to a variable, the literal value null is used to show this. You may use it with any NON-primitive variable.

Automatic Initalizations

- Field variables are those that are declared as members of a class.
- Field variables and array elements are always initialized to their default values automatically, unless they are of object types, in which case they are initialized to null.
- Local variables (automatic variables) are declared local to a method.
- Local variables are NOT initialized automatically. You will receive a compile error when trying to use a local variable that has not been initialized.

The following are the default values for each of the primitive types:

Type	Default Value
boolean	False
byte	0
char	\u0000
short	0
int	0
long	0l
float	0.0f
double	0.0d

© 2001 All Rights Reserved – BrainBuzz.com

Operators and Assignments

Urnary Arithmetic Operators

Operator	EX	Result
+	+77	value of 77
-	-35	negative value of 35
++(Prefix)	a = 12 b = ++a	b has the value of 13
++(Postfix)	a = 12 b = a++	b has the value of 12
--(Prefix)	a = 12 b = --a	b has the value of 11
--(Postfix)	a = 12 b = a--	b has the value of 12

Binary Arithmetic Operators

+	12 + 8	value of 20
-	99 – 90	value of 9
*	6 * 10	value of 60
/	13/6	Div: value of 2
%	13%6	mod: value of 1

Numeric promotion

Numeric promotion is always performed when binary operations are used on integer and floating point numbers. Here are the rules to remember:

- If one of the operands is a `double`, then the other operand is converted to a `double`.
- Otherwise, if one of the operands is a `float`, then the other operand is converted to a `float`.
- Otherwise, if one of the operands is a `long`, then the other operand is converted to a `long`.
- Otherwise, both operands are converted to `int`.

When one of the operands is a String in a '+' operation, then the other operand is converted to a String object.

246.8 + "99" results in "246.899"

© 2001 All Rights Reserved – BrainBuzz.com

Division

Java handles division by zero in different ways depending on if it is performed on integers, positive floating point numbers, or negative floating point numbers.

- Division of an integer by zero results in an ArithmeticException
- Division of a positive floating point number by zero results in POSITIVE_INFINITY
- Division of a negative floating point number by zero results in NEGATIVE_INFINITY

Modulo

- The modulo (%) operator results in the remainder when one operand is divided by the second. The following formula is how it works on **x % y**:
  x – ((int) (x/y)*y)
- When using the modulo operator, remember that Java allows floating point operands.
- If x is positive, the result will ALWAYS be positive (even if y is negative).

Runtime Exceptions

An ArithmeticException is thrown when dividing by zero or using the modulo operator on a zero during integer operations, and is the ONLY runtime exception that can occur during arithmetic operations.

Overflow and Underflow

Java handles overflow and underflow problems by discarding any high-order bytes that will not fit into the allowable number of bytes.

String Operators

Operator	Example	Result
+	"a"+"c"	"ac"

Equality Operators

- Used with all 8 primitives, as well as object references.
- Return a boolean value in all cases.

Operator	EX	Result
<	9 < 100	true
<==	3 <= 9	true
>	9 > 100	false
>==	3 >= 9	false

© 2001 All Rights Reserved – BrainBuzz.com

==	3 == 9	false
!=	3 != 9	true
instanceof	"Hello" instanceof String	true

When using primitives, remember:

- A numeric value can not be compared to a boolean value (compile error results).
- Numeric promotion takes place before any numeric values are compared.

- When using the equality operators to compare Object references, the == and != operators will check to see if the objects are the same instance -- **NOT** if they have the same values.
- The equals() method of the Object class allows you to tell if two objects have the same value.

Consider this table showing two situations using the == operator and equals():

		s1 == s2	**s1.equals(s2)**
A	`String s1 = "Java";` `String s2 = "Java";`	true	true

B	`String s = "ja";`
	`String s1 = s + "va";`
	`String s2 = "java";`
		false	true

- In situation A, the evaluation will always return **true**, since there is only one String object created (s1 and s2 are both pointing to the same one).
- To avoid the above situation, use the *new* keyword so that s1 and s2 are created as different objects.
- In situation B, s1 and s2 are pointing to different objects, because s1 is created at runtime.

<u>Logical Operators</u>

There are two groups of logical operators in Java:

- the boolean operators
  - & (and)
  - | (or)
  - ! (not)
  - ^ (exclusive-or)
- and the logical short-circuit operators
  - && (short-circuit and)
  - || (short-circuit or)

© 2001 All Rights Reserved – BrainBuzz.com

CramSession Prepare for Success!

- The && short-circuit operator does not evaluate the 2nd term if the 1st term is evaluated to false.
- The || short-circuit operator does not evaluate the 2nd term if the 1st term is evaluated to true.

Truth Table for the Logical Operators

Op1	**Operator**	**Op2**	**Result**
false	&	false	false
false	&	true	false
true	&	false	false
true	&	true	true
false	|	false	false
false	|	true	true
true	|	false	true
true	|	true	true
false	^	false	false
false	^	true	true
true	^	false	true
true	^	true	false
false	&&	false	false
false	&&	true	false
true	&&	false	false
true	&&	true	true
false	||	false	false
false	||	true	true
true	||	false	true
true	||	true	true

Assignment Operators

- The basic = operator is used on its own to assign values (x = 99), or in combination with other binary operators to "short-cut" assignments (op=).
  x += y   x is assigned the result of x + y

  a /= b   a is assigned the result of a / b
- It is important to know that when using this form of op= assignment, the result is cast to the type of the left hand operand. But when you use the normal method (x = x + y), NO cast is done.
- The assignment operator is the ONLY right associative operator.
  x = y = z is evaluated as x = (y = z)

© 2001 All Rights Reserved – BrainBuzz.com

The Ternary Operator

This operator, ? : , is used in the following manner:
operand1 ? operand2 : operand3

- operand1 must of type, boolean.
- If operand1 is true, then operand2 is returned.
- if operand1 is false, then operand3 is returned.

The Cast Operator

- Done by placing the destination cast type keyword between parentheses before the source type expression.
- It is used to convert values of one type to another, or to change an object's reference type to one that is compatible.
- When casting "down" to a smaller numeric type (double to an int), you need to explicitly use the cast operator, but be aware that you can lose data.
- When casting "up" to a higher numeric type, you do not need to explicitly use the cast operator.
- Consider the following 2 examples:

```
1  public class example1{
2        public static void main (String [] args){
3              int x = 99;
4              double y = 5.77;
5              x = (int)y;
6
7              System.out.println("x = "+ x);
8         }
   }
```

In the above example, the integer, x, is being set to the value of the double, y. This is a case of a narrowing conversion, where a larger type is being 'forced' into a smaller type. This results in loss of data. The result printed to the screen is 5.
Also, notice that in line 5, there must be an explicit cast to the int, as without it, a compiler error would be generated.

```
1  public class example2{
2        public static void main (String [] args){
3              int x = 99;
4              double y = 5.77;
5              y = x;
6
7              System.out.println("y = "+ y);
8
```


© 2001 All Rights Reserved – BrainBuzz.com

CramSession Prepare for Success!

```
        }
}
```

In this example, the double, y, is being set to the value of the integer, x. This is a case of a widening conversion, where a smaller type is set to a larger type. The result printed to the screen is 99.0.
Also, notice that in line 5, there is no explicit cast required.

Bitwise Operators

- There are four bitwise operators, but you will not need to know the unary ~ (inversion) operator for the exam.
- The other 3 are binary operators. These operators all work in a bit-by-bit method. Here are some rules to remember:

& (and)	returns a 1 bit if both operands are a 1, otherwise it returns 0.
| (or)	returns a 0 bit if both operands are a 0, otherwise it returns 1.
^ (exclusive-or)	returns a 0 bit if both operands are a 1 or both are a 0, otherwise it returns 1.

Shift Operators

These binary operators also work at the bit level.

>>(right shift)	the bits of the left operand are shifted to the right based on the right operand value. The value of the leftmost bit before the operation is what fills in the new space on the left hand side. This bit also represents the sign.
<< (left shift)	the bits of the right operand are shifted to the right based on the right operand value. O's are used to fill in the right side.
>>>(unsigned right shift)	same as >>, except 0's fill in the left side.

The instanceof Operator

- This binary operator is used to determine if an object reference (left operand) is an instance of the class, interface or array type of the right operand.
- returns a boolean value.

© 2001 All Rights Reserved – BrainBuzz.com

Declarations & Access Control

Declaring Field and Local Variables

There are two types of variables in Java, both declared using the syntax,
`modifier type identifier`
e.g. `private int numBooks;`

field variables – declared as members of a class.
local variables – declared local to a method.

- The type can be a primitive, object or an array.
- The modifier determines access – public, protected and private, final, static, transient, and volatile.

*Note: Local variables can only use the final modifier.

Declaring Arrays

- Arrays are declared using the square brackets "[]".
- Remember that they are Java objects.
- The following are legal declarations:

```
int[] x;

int x[];

int i[][]; //declares a two dimensional array.

int[  ] x;
```

There are two ways to create an array:

1. create the array by declaring it and initializing its elements.
   `int[]myarray = {21,24,35,49,54,78}`

   This creates a 6 element array with the given int values.

2. use the *new* keyword when creating large arrays.
   `int[]myarray = new int[10];`
   This creates a new 10 element array holding *int* types, but no values yet.

   A *for* loop may be used to initialize the values of each element.

```
for(int i=0; i < myarray.length; i++){
    myarray[i] = i;
}
```

Arrays start counting from element 0 (zero).

© 2001 All Rights Reserved – BrainBuzz.com

Method Declarations

Declared using the syntax:

```
modifier returnType nameOfMethod(parameters) throws clause{
//body
}
```

- modifier may be : public, private, protected, default(package access), abstract, final, native, static, or synchronized.
- returnType may be: void, primitive, or an Object.
- parameters is a comma-separated list of *type typename* combinations. You can also declare a parameter as final, so that a local inner class can have access to it.
- The throws clause identifies the exceptions that this method may throw but does not catch during the methods execution. The type of the throws clause must be a superclass of the exception.

- Static methods are ones that belong to the class as a whole, and so do not require an instance of the class.
- When primitives are passed into a method, they are passed by value, that is, a copy of the value is made. This means that the original is NOT modified.
- When passing in an object as a parameter, a copy of the reference to the object is used, and not the actual object (passed by reference). This means that any changes to the reference within the method do NOT affect the reference to the original. But, any changes to that object itself in the method, do affect the original.

Consider these two examples illustrating the difference between passing by value and passing by reference:

A - Passing a primitive to a method (pass by value)

```
public class PassByValue{
      public static void main (String [] args){
            int x = 0;
            setThis(x);
            System.out.println("x ==> " + x);
       }

      static void setThis(int a){
            a = 10;
      }
 }
```

The result printed out is 0. This is because a

© 2001 All Rights Reserved – BrainBuzz.com

```
copy was made and sent to the method,
setThis(). Therefore, setting the value to 10
in the method did not affect the original value
of x in the main.
```

B - Passing an object to a method (pass by reference)

```
public class PassByRef{
      public static void main (String [] args){
            int x = 0;
            int temp[] = new int[1];
            temp[0] = x;
            setThis(temp);
            x = temp[0];

            System.out.println("x ==> " + x);

       }

      static void setThis(int a[]){
            a[0] = 10;
      }
}
The result printed out is 10, since a reference
to the temp array was used to send to the
method, setThis(). Thus, setting the value to
10 in the method, did indeed affect the
original value of x in the main. Don't forget
that arrays are objects!
```

Access Modifiers

Access modifiers provide the way to provide encapsulation (data-hiding) in Java.

public

May be accessed outside of its package.

private

- May be accessed only from within its class.
- Not applied to classes.

protected

May be accessed only by classes in the same package or subclasses of this class.

(none)

May be accessed only by classes in the same package.

© 2001 All Rights Reserved – BrainBuzz.com

Other Modifiers

static

- Applies to the class, not to any particular instance of the class.
- When used with variables, there is only one copy for all instances of the class. If an instance changes the value, the other instances see the changes.
- When used with methods, it can be called without having created an instance. A static method may NOT refer to non-static variables without using an instance of the class. You may not use the *this* keyword when referring to the instance.

final

- When used with variables, it indicates that a variable may not be changed.
- When used with methods, it prevents a method from being overridden.
- When used with classes, it prevents a class from being subclassed.

abstract

- Identifies an abstract class, that defers its implementation to its subclasses.
- It may not be instantiated.
- Abstract classes may declare abstract methods that also defer implementation. A semi-colon replaces the body of the method.

native

- Identifies a method written in a non java language.
- Outside the JVM in a library.

synchronized

indicates that only one thread at a time may access this method for a certain object or class.

transient

- Used with variables to indicate that a variable may not be serialized.
- It may not combined with either the final or static modifiers.

© 2001 All Rights Reserved – BrainBuzz.com

volatile (not covered on exam)

- Used with variables.
- Indicates that a variable may be accessed by unsynchronized threads

Summary of Access Modifiers

Accessible to	public	protected	package	private
Same class	yes	yes	yes	yes
Different Class but same package	yes	yes	yes	no
Subclass in different package	yes	yes	no	no
Non-subclass in different package	yes	no	no	no

Flow Control

For the following flow control statements, we will show examples to demonstrate format.

if/else

```
if (x = 100) {
        System.out.println("x is 100");
}
else {
        System.out.println("x is NOT 100");
}
```

switch/case

```
int x = 2;
switch (x){
        case 1:
                System.out.println("x = 1");
                break;
        case 2:
                System.out.println("x = 2");
                break;
        case 3:
                System.out.println("x = 3");
                break;
        default:
```

© 2001 All Rights Reserved – BrainBuzz.com

```
            System.out.println("default");
}
```

- The break statement is optional, and is used in each case to "break" out of the switch, so that the execution will not continue.

for

```
for (x = 0; x < 10; x++){
      System.out.println(x);
}
```

while

```
while (x < 100){
      x += y;
}
```

do/while

```
do{
      x += y;
}while (x < 100)
```

break/continue

- *break* causes the current loop to be abandoned.
- *continue* causes execution to skip the rest of the code in the current iteration and start at the top of the loop with the next iteration.
- Labeled versions of *break/continue* allow you to jump to/break out of, wherever the label is. A label is written using an identifier followed by a colon(:). The label encloses the code with curly brackets.
- Remember that Java does NOT support *GOTO*.

return

- Use *return* to exit from the current method, and jump back to the statement of the calling method.
- *return* can be used to return a value from a method that is declared to return void.
- The value returned must match the type of method's declared return value.

© 2001 All Rights Reserved – BrainBuzz.com

Exception Handling

- The Java runtime system will "throw" exceptions that you may then "catch" and handle gracefully in some fashion.
- You may also "throw" your own exceptions (called checked exceptions), to avoid a potential upcoming error. This is done using the *throw* statement and the *new* keyword.
- Methods may have a *throws* statement in their declaration.
- Runtime exceptions are objects of `java.lang.RuntimeException`, `java.lang.Error`, or of one of their sub-classes.
- A *try* statement (and curly braces) are used to enclose a block of code where an exception may occur. Then, a *catch* clause(s) is used to handle each of the different types of exceptions. An optional, *finally* clause may be placed at the end, and is executed always.
- You may have several catch clauses to catch different types of exceptions. It is a good idea to always catch a general Exception as your last *catch* clause. The more specific ones should be before it.
- The following example shows the proper format using the *try, catch* and *finally*.

```
1       public class t1{
            public static void main (String [] args){
  2           String s = new String("Java");
  3           try{
  4
  5                 System.out.println(s.charAt(8));
  6           }
  7           catch(IndexOutOfBoundsException e){
  8                 System.out.println("IndexOutOfBounds Exception");
  9           }
 10           catch(Exception e){
 11                 System.out.println("General Exception");
 12           }
 13           finally{
 14                 System.out.println("Always gets printed out");
 15           }
 16         }
 17       }
```

In this example, line 5 will throw an IndexOutOfBoundsException, and will be caught in the specific catch clause, printing out "IndexOutOfBounds Exception". The finally clause will also print out "Always gets printed out".

© 2001 All Rights Reserved – BrainBuzz.com

Overloading, Overriding, Runtime Type and Object Orientation

Object Orientation

Inheritance

Allows objects to be created from other objects by extending them. This passes on the same behaviors and attributes of the superclass to the subclass, allowing the sub-class to then build in more functionality as required.

- Java only supports single inheritance.
- Constructors are not inherited in Java.

Encapsulation

The important OO feature of data-hiding. That is, controlling the access to details of an object's implementation. The data in a well encapsulated object is protected from being used improperly or accidentally. A fully encapsulated object will only have private members that are accessible with get() and set() methods.

- In Java, you use access modifiers to promote encapsulation.

Polymorphism

This object-oriented principle says that a method will behave in a different manner when performed on different classes. This means that subclasses may have methods with the same name as their superclass, but perform differently to achieve a similar result. Here is an example:

> class TaxableObject has a calculateTax() method.
>
> > class House extends TaxableObject.
> > class Cottage extends TaxableObject.
> > class Boat extends taxableObject.
>
> The three classes that extend TaxableObject, also have their own specific versions of calculateTax(), since they all require different calculations to figure them out. In this way, calculateTax(), is said to exhibit polymorphism.

Java promotes polymorphism using inheritance and overridden methods.

© 2001 All Rights Reserved – BrainBuzz.com

Designing Classes with "is a" and "has a" Relationships

Consider the following example:

```
Public class Employee extends Person{

      String employeeNum;
      double salary;
}
```

This says that an employee "is a" person, and "has a" salary and an employee number.

Overloading Methods

When you **overload** a method, you use the same name, but different arguments and return types. The Java compiler can differentiate between two overloaded methods with the exact same name, because they have different signatures.

Overridden Methods

- Subclasses inherit the non-private members of its superclass. In the subclass, you may want to **override** a superclass method to provide a different sort of functionality. The name, arguments, and return type are the same in overridden methods.
- When overriding methods, you may not use access modifiers that may make the method more restrictive than the overridden one in the superclass.
- Also, the throws clause (if present), may only throw exceptions that the overridden method in the superclass throws.

Constructors

- Remember that constructors are NOT inherited.
- You can declare constructors private if you do not want a class to be instantiated.
- You may use this() and/or super() to access overloaded or parent-class constructors. You must place them at the very beginning of the code block in the constructor, and you may only make one of these types of calls.
- *this()* is used to call any of the other overloaded constructors defined in the class before performing actions specific to current constructor.

```
public class Book{

      String title, author;
      boolean inStock;
      public Book(String title,String author,boolean inStock){
            this.title = title;

            this.author = author;
            this. inStock = inStock;
```


© 2001 All Rights Reserved – BrainBuzz.com

```
        }
        public Book(String title){
              this(title, "No author name supplied", false);
        }
        public Book(){
              this("No title supplied");
        }
}
```

```
Here, class Book has three constructors allowed to create a new
Book object. The first one takes all three required arguments.
The second one takes only a title argument, and then uses the
this() method to call the constructor that takes the three
arguments, passing in default values. The last constructor,
creates a Book object without any argument. It uses the this()
method with a default value for title, and then calls the
constructor that takes one argument, allowing it to fill in the
rest of the object's values.
```

In the above example, if Book extended some other class, you could use the super() method to call the constructor of the superclass to create the objects.

Inner Classes

- A class has variables and methods as members, but it may also declare a class as a member. This inner, or nested, class has the same access as the variables and methods.
- A **non-static inner class** may not be created before first creating an instance on its outer (top level) class.
  if the code fragment, `Outer x = new Outer();` creates an instance of the outer class, then to create an instance of the inner class, you could use: `Inner y = x.new Inner();`
  A shorter form could also be used here to create them both objects in one line of code in Outer: `new Outer().new Inner();`
- A **static inner class** does not require an instance of its surrounding class to be instantiated first, and it may not refer to instance variables and methods of the containing class directly.

Anonymous Classes

- An anonymous class is defined by instantiating it "on the fly" inside a method. It is basically implementing an interface or extending a class without using the *implements* or *extends* keywords.
- They do not have constructors.

© 2001 All Rights Reserved – BrainBuzz.com

- They are useful when implementing event listeners or extending adapter classes.
  This example uses an anonymous class to create an ActionListener object as an argument to the addActionListener() method.

```
Public class SomeClass extends Frame{
      Button b = new Button("Click ME!");
      b.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
   System.out.println("You pressed " + e.getActionCommand());
            }
      });
}
```

Garbage Collection

- When your program has no more references to an object, and it is not possible to be used again, it becomes unreachable, and is then subject to garbage collection.
- The garbage collector gives the object a chance to clean up after itself before it is collected, by invoking the object's finalize() method.
- Garbage collection is not determinable. There is no way to know when an object will be collected.
- You can increase the likelihood of object finalization and garbage collection using the System class's runFinalization() method, which in turn will call the finalize() methods on all objects that are waiting to be garbage collected.

Threads

In Java, threads may be created in one of two ways:

- using the first method, you create a class that subclasses Thread, and then override the run() method. When an instance of the new class is created, a call to its start() method is made, automatically invoking the run() method.

```
class MyThreadClass1 extends Thread{
      public static void main (String args[]){
            MyThreadClass1 t = new MyThreadClass();
            t.start();
      }
      public void run(){
            //code
      }
}
```

© 2001 All Rights Reserved – BrainBuzz.com

The other method to create a thread involves implementing the java.lang.Runnable interface, which has one method, run() that you must override. To create a thread using this method, you must create a new Thread object, passing in a new object of your class (that implements Runnable) as an argument. Then calling the start() method will invoke the run() method automatically.

```
class ThreadEx2{

      public static void main (String args[]){
            Thread t = new Thread(new MyThreadClass2());

            t.start();
      }
}
class MyThreadClass2 implements Runnable{
      //variables, constructors, etc.
      public void run(){
            //code

      }
}
```

A thread may be in any one of the following states:

Ready – the thread is created but is not yet "started".

Running – the thread is executing.

Waiting – the thread is currently not executing. A thread may enter the waiting state if the sleep() or wait() method is invoked. It may also enter the waiting state, if the thread is blocking on I/O (disk writes, etc), or is unsuccessfully attempting to acquire a lock on an object.

Dead – the thread has completed its processing. A dead thread may not be restarted.

The following diagram illustrates the ONLY valid thread state transitions:

In other words, a thread may not, for example, go from a Ready state to a Waiting state.

© 2001 All Rights Reserved – BrainBuzz.com

- NOTE: The stop(), suspend(), and resume() methods are deprecated for JDK 1.2.
- A thread's priority tells the thread scheduler when this thread can be run in relation to other threads. There are integer constants defined in the Thread class, ranging from MIN_PRIOTITY to MAX_PRIORITY, with the default, being NORM_PRIORITY.
- Java provides a way to coordinate control to shared data or resources (i.e. a database record). The use of synchronized methods and statements provide a mechanism to get "locks" on certain objects that require controlled access.
- The keyword, *synchronized*, is used to synchronize a method.

```
public static synchronized void someMethod(){

}
```

- Invoking a synchronized method on an object, prevents other synchronized methods from accessing that object until the method has completed executing. A synchronized method is used with its object's or class's lock.
- You may also use a synchronized statement, using enclosing curly braces around a section of code.

```
synchronized(someObject){
//statements to execute

}
```

The statements within the curly braces will only be executed when the thread acquires a lock for the object enclosed in the ().

- The wait(), notify(), and notifyAll() methods give a programmer better control to synchronize execution. Threads that are "waiting" for the right condition in order to execute, may be "notified" by another thread that has acquired the object's lock, returning the waiting thread to the Ready state.
- Invoking the yield() method will cause a thread to enter its Ready state.
- Invoking the sleep() method will cause a thread to enter its Waiting state.

The java.lang package

For the exam, you must be familiar with a number of methods from the Math class, as well as have a good understanding of the String and StringBuffer classes. This is a good time to look through the API for specifics on the methods. The Sun API is here: http://java.sun.com/j2se/1.3/docs/api/index.html

© 2001 All Rights Reserved – BrainBuzz.com

The Math class

abs(arg a)	returns the absolute value of the double, float, int or long arg.
ceil(double a)	returns the smallest double value that is not less than the argument.
floor(double a)	returns the largest double value that is not greater than the argument.
max(arg1 a, arg2 b)	returns the greater of the two values (double, float, int or long args).
min(arg1 a,arg2 a)	returns the smaller of the two values (double, float, int or long args).
Random()	returns a random number between 0.0 and 1.0.
round(arg a)	returns the closest to the long or double arg.
sqrt(double a)	returns the square root of the double argument.
cos(double a)	returns a double cosine of an angle.
tan(double a)	returns a double tangent of an angle.
sin(double a)	returns a double sine of an angle.

The String Class

- It's a good idea to familiarize yourself with the different methods and constructors in these classes using the API.
- In Java, a String may not be modified (immutable), and so may not be changed once it is created, while a StringBuffer object may grow and may be modified (mutable).
- Some of examples of methods of the String class:

length()	gets the string length.
toUpperCase()/toLowerCase()	converts string to uppercase/lowercase.
equalsIgnoreCase(String)	compares the content of two string objects while ignoring the case.
charAt(int)	returns the character at the specified index.
concat(String)	concatenates the specified string to the end of the current string.
substring(int)	returns a new string that is a substring of the current string.
trim()	removes all white space from both ends of the string.
toString()	returns a string representation of the object.

© 2001 All Rights Reserved – BrainBuzz.com

StringBuffer Class

A StringBuffer object may be modified(mutable). When you are using the binary String concatenation operator (+), behind the scenes, the Java compiler is actually using StringBuffers for the implementation.

```
s = "z" + 9 + "y";
```

The compiler performs this operation something like this:

```
s = new
StringBuffer().append("z").append(9).append("y").toString();
```

There are 3 constructors for the StringBuffer class:

`StringBuffer()`	a string buffer with no characters is created with an initial capacity of 16 characters.
`StringBuffer(int length)`	a string buffer with no characters is created with an initial capacity of the specified length.
`StringBuffer(String s)`	a string buffer is created from the specified String object.

The most common methods are the different versions of the append() method, and the insert() methods. Again, be sure to check the API and go over the different methods and constructors.

The java.io Package

As with any of the packages you are required to know for the exam, you should make sure to go through the API, but there are some main topics you can concentrate on here.

File Class

Using an operating system's own convention for naming files, this class provides the ability to manipulate file and directory names. Once a File object is created, you have access to a number of methods allowing you to create, rename, and delete files, as well as access to pathnames, and read and write permissions.
Here is a small example showing some of the basic methods:

```
1  import java.io.*;
   public class filetest{
2        public static void main (String [] args){
3              File f = new File("C:\\Program Files\\");
4
5              if (f.isDirectory()){
6                    File[] fa = f.listFiles();
```


© 2001 All Rights Reserved – BrainBuzz.com

CramSession Prepare for Success!

```
7               }
8               File[] yourRoots = f.listRoots();
9      String fullPathname = f.getAbsolutePath();
10
11     String parentDir =  f.getParent();
1
 2
           }
    }
```

- Line 4 creates a new File object with an argument representing the full path to the directory of file. Note, this is only one of the three constructors.
- Line 5 tests to see if the object is a directory. You can also test if it is a file.
- The listFiles() method in line 6, returns an array of File objects representing the directories and files.
- In line 8, the listRoots method returns an array of File objects that represent all of the system's root directories.
- Lines 9 and 10 both get String objects returned, representing the absolute path, and the parent directory of the File object, respectively.

InputStreamReader and OutputStreamWriter Classes

- These classes are useful when the default character encoding of the platform is not sufficient, and you need to change it in order to communicate with a different type of character encoding depending on the locale/language and the platform.
- The **InputStreamReader** class converts an object from **a byte stream → character stream**. Two constructors are provided. One takes an InputStream object only, and uses the default character encoding. The other constructor also takes a String representing the encoding to use.
- The **OutputStreamWriter** class converts an object from **a character stream → byte stream**. The two constructors take similar arguments as the InputStreamReader, except the first argument is an OutputStream object.
- The getEncoding() methods of each class will return a String representing the encoding used by the stream.

© 2001 All Rights Reserved – BrainBuzz.com

FilterInputStream and FilterOutputStream

These two classes provide the capability for you to take a stream and create another one, allowing you to use the output of a previous filter as an argument for the current one. This method is known as "chaining", and is useful when working with I/O streams.

FileInputStream and FileOutputStream

The FileInputStream class gives you the ability to get input from a file once an object of this type is created, by using one of its three constructors, and taking either a File object, a FileDescriptor object or a String representing the path to the file.
The FileOutputStream class allows you to open an existing file or create a new one, and then output a stream of data. It has similar constructors as the FileInputStream, with an additional one that takes a String and a boolean, allowing you to append to the file, instead of overwriting it.
Here is an example using these two classes:

```
import java.io.*;
public class FileInOut{
      public static void main (String [] args){
            //first create an output file
            try{
   FileOutputStream fos = new

   FileOutputStream("C:\\fileTest.txt");


   String text = "May the force be with you.";


            //Now write this string to the new file

            for (int i=0;i<text.length();i++){
                  fos.write(text.charAt(i));
            }
            fos.close();

   /*Now let's make sure our text is there is in there*/


            //First open for reading.
   FileInputStream fis = new

   FileInputStream("C:\\fileTest.txt");


            /*one of the read() methods takes a byte
            array and returns an int representing the
```


© 2001 All Rights Reserved – BrainBuzz.com

```
            number of bytes to read.*/
            byte data[] = new byte[fis.available()];
            int numBytes = fis.read(data);
  System.out.println("There were " + numBytes + " bytes

  to be read in");


  System.out.println("The sentence says :  " + new

  String(data));


            fis.close();
              }
      catch(IOException ioe){}
       }
}
```

RandomAccessFile Class

- This class is different than the other reading and writing classes in the java.io packages, because a RandomAccessFile object can do BOTH reading and writing of data, and to any part of a file.
- These objects make use of a filepointer to keep track of its position in the array of bytes. The seek() method allows you to set the filepointer somewhere in the file, and the getFilePointer() method returns the current position of the filepointer. There are two constructors, both taking a String, and they represent the mode ("r" or "rw"), as well as either a String file name, or a File object.

The java.util Package

The new Collections API has been added for JDK 1.2. They provide classes and interfaces to work with collections of objects.
Here is the hierarchy of these interfaces and classes:

Interfaces

```
Enumeration
Collection
        List
        Set
    SortedSet

Comparator
```

© 2001 All Rights Reserved – BrainBuzz.com

Iterator

ListIterator

Map

SortedMap

Map.Entry

Classes

AbstractCollection
- AbstractList
  - AbstractSequentialList
    - LinkedList
  - ArrayList
  - Vector
    - Stack
  - AbstractSet
    - HashSet
    - TreeSet
- AbstractMap
  - HashMap
  - TreeMap
  - WeakHashMap
- Arrays
- BitSet
- Collections
- Dictionary
  - HashTable
    - Properties

The original Collections (and a brief description) from JDK 1.1:

Vector	An array of objects that may grow and shrink.
Hashtable	An object that uses a key-value system to store and retrieve objects.
Stack	Extends Vector, and stores objects in a last-in/first-out stack.
Enumeration	This interface gives you the ability to iterate through the objects in a Vector, Stack, or Hashtable. It is now being replaced with the Iterator Interface for JDK 1.2.

© 2001 All Rights Reserved – BrainBuzz.com

The new Collections Interfaces

Collection	The root of the Collections hierarchy. This is where you will find implementations to work with groups of objects.
List	Also holds objects in an ordered collection, but allows duplicates. The Vector class implements this interface.
Set	Holds objects in a fixed order. No duplicates are allowed, and one element may have a null value.
SortedSet	Extends Set, with elements that are sorted in an ascending order.
Map	Provides key-value functionality. Replaced the Dictionary class from JDK 1.1.
SortedMap	Extends Map, and provides elements that are sorted in ascending key order.
Iterator	Provides functionality to iterate through a collection of objects. Replaced the Enumeration interface for JDK 1.2.
Comparator	Allows comparisons to be made of the elements in a collection.

And here are a few, but not all, of the Collections classes:

ArrayList	Provides implementation of an expandable array. It is similar to a Vector, but is somewhat faster, as it does not use synchronized methods.
HashSet	A key-value pair implementation, which allows null elements.
LinkedList	A linked list implementation, which allows null elements.
Collections	Not to be confused with the Collection class, this one provide static methods for operating on objects implementing the Collection interface.

There are also five Abstract classes that provide a "bare bones" implementation of many of the methods.

The java.awt Package

The Abstract Window Toolkit provides the functionality to development GUIs.

Components and Containers

- The framework for a GUI application is provided by the java.awt.Component class and its subclasses. Every GUI element, except for menu components, are subclasses of Component. Menus are subclasses of the java.awt.MenuComponent class.
- **Components** are the visible, graphical elements of the GUI – such as buttons, checkboxes, labels, scrollbars, etc. **Containers** are components that "hold" other components. They include panels, windows, frames, etc.
- The Component class has, among others, size, font, and color attributes. There are over 100 methods of the Component class that you should somewhat familiarize yourself with. There are set and get methods for properties like: bounds, size, font, background, foreground, and color. You can also get a component's container with the getParent() method. To get a thorough familiarity with these methods, you should use the API.

© 2001 All Rights Reserved – BrainBuzz.com

- The Container class supplies methods to add, remove, locate (get), and position its components, contained within. The three main subclasses of Container are Window, Panel, and ScrollPane.

Layout Managers

- Containers use layout managers to manipulate where the components are positioned. These layout managers implement layout policies that provide rules for organization.
- To set a layoutManager, a container object will call its setLayout() method that takes a LayoutManager object as its argument.

```
ScrollPane s = new ScrollPane();
s.setLayout(new BorderLayout());
```

- Some containers even have a constructor that takes a LayoutManger object as its argument, in which case you could do this:
  `Panel p = new Panel(new GridBagLayout());`

There are five LayoutManager classes:

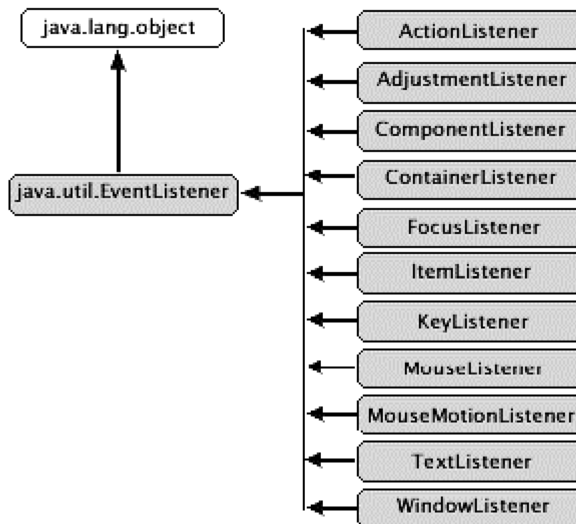
BorderLayout	Uses the North, South, West, East and Center to position components
FlowLayout	Lays components out from left to right, top to bottom fashion. Each component is given its "preferred size".
CardLayout	One component is visible at a time. Think of them as being in a stack of cards, and you can "flip" to the first, last, next, or previous ones.
GridLayout	Components are laid out in a grid with each component being the same size.
GridBagLayout	This one is similar to the GridLayout, but more flexible, in the way that some components may fill more than one row or column. Each component is associated with a GridBagConstraints object, which has detailed parameters for you to access.

Event Handling

- With the JDK 1.2 came a new event model to replace 1.02's *inheritance model*, which had events "bubbling" up to an object's container when it could not handle the event. The more efficient, and easier to use, *event delegation model*, of 1.2, uses event listeners to listen for the generation of events that they have been registered to listen for, and handles them accordingly.

© 2001 All Rights Reserved – BrainBuzz.com

- The java.util.EventListener is the top level interface for event listeners. The class you create to handle the events will implement one or more of these interfaces (shown in gray):



This is how events are handled:

- A control (component) on a GUI will generate an event when the user clicks it. Zero or more eventListeners may be "registered" with that control.
- For each registered listener of the appropriate type, the event handling method of the listener is invoked automatically to process the event generated by the user clicking the button.

So to process an event, as a programmer, you must perform these two tasks:

1 – register an event listener.
2 – implement an event handler

Event Listener – an object of a class that implements one or more event-listener interfaces.

Event Handler – a method that is automatically invoked in response to a certain type of event.

Here are details of a button click, and how you handle the event it generates:

- A button generates an **ActionEvent**.
- An action event may be handled by any **ActionListener**

© 2001 All Rights Reserved – BrainBuzz.com

CramSession Prepare for Success!

object (a class that implements the ActionListener interface). Notice that ActionListeners listen for ActionEvents.

- When you implement the ActionListener interface, you are forced to provide a definition for the **actionPerformed()** method. Once you register an object to handle an ActionEvent, the actionPerformed() method is called whenever an actionEvent occurs. The actionEvent object is sent to the actionPerformed() method, and contains specific information about the event that just occurred.

The following are the steps necessary to register a listener to handle a button click, followed by a sample of code:

1. Create a class that implements the ActionListener interface.
2. Then override the actionPerformed() method for this class.
3. Register an instance of this class with the button.

In the code below, these three steps are numbered in the left-hand column.

```
  | import java.awt.*;
  | import java.awt.event.*;
  | public class BigButton extends Frame{
  |
  |         Button b;
  |         public BigButton(){
  |                 super("Our Button Event Tester");
  |                 setupFrame();
  |         }
  |         public void setupFrame(){
  |                 setSize(300,100);
  |                 Panel p = new Panel();
  |                 add(p);
3 |                 Button b = new Button("Hello");
  |                 b.addActionListener(new ClickMeHandler());
  |                 p.add(b);
  |                 Label l = new Label("");
  |                 p.add(l);
  |                 //add a handler to close frame.
  |                 addWindowListener(new WindowAdapter(){
  |                         public void windowClosing(WindowEvent we){
  |                                 System.exit(0);
  |                         }
  |                 });
  |                 show();
  |         }
  |         public static void main(String args[]){
  |                 BigButton ob = new BigButton();
  |         }
1 | }
2 | class ClickMeHandler implements ActionListener{
  |         public void actionPerformed(ActionEvent e){
  |                 Button theButton = (Button)(e.getSource());
```


© 2001 All Rights Reserved – BrainBuzz.com

```
                String theText = theButton.getLabel();
                if (theText.equals("Hello")){
                        theButton.setLabel("World");
                }
                else{
                        theButton.setLabel("Hello");
                }
        }
}
```

Special thanks to
Trent Heintz
for contributing this
Cramsession.

© 2001 All Rights Reserved – BrainBuzz.com